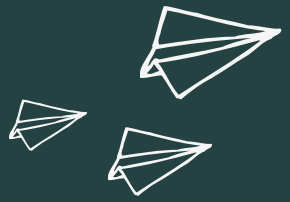


go 语 言 编 程

目录页

第七章
Part seven

1 并发编程的相关概念

2 goroutine

3 channel

4 共享内存

第七章 Go语言并发编程

7.1 相关概念

1 进程和线程

在现代操作系统中，线程是CPU调度和分配的基本单位，进程则作为资源拥有的基本单位。每个进程是由私有的虚拟地址空间、代码、数据和其它各种系统资源组成。线程是进程内部的一个执行单元。 每一个进程至少有一个主线程，这个主线程无需由用户去主动创建，是由系统自动创建的。 用户根据需要在应用程序中创建其它线程，多个线程并发地运行于同一个进程中。

第七章 Go语言并发编程

7.1 相关概念

2 并发和并行

并发与并行（Concurrency and Parallelism）是两个不同的概念，理解它们对于理解多线程模型非常重要。并发是指在一个时间段内有多个线程或进程在执行，但在某个时间点上只有一个在执行，多个线程或进程通过争抢CPU时间片轮流执行。并行是指一个任意的时间点上都有多个线程或进程在执行。

并发就像一个家长(cpu)在喂多个孩子(线程)，轮换着每个孩子喂一口，表面上多个孩子都在吃饭。并行就像n个家长(cpu)在喂n个孩子(线程)，这n个孩子同时都在吃饭。并行需要硬件支持，单核处理器只能是并发，多核处理器才能做到并行。

7.1 相关概念

3 多线程和多核CPU

多核处理器是指在一个CPU处理器上集成多个运算核心从而提高计算能力，也就是有多个真正并行计算的处理核心，一般一个处理核心对应一个内核线程。
程序一般不会直接去使用内核线程，而是使用用户线程。用户线程与内核线程的对应关系有三种模型：一对一模型、多对一模型、多对多模型。

1> 一对一模型（1:1），一个用户线程对应一个内核线程。
优点是线程之间并行处理，一个线程阻塞，不会影响其它线程。缺点是操作系统内核线程调度时，上下文切换的开销较大，导致用户线程的执行效率低。

7.1 相关概念

2> 多对一模型（N:1），多个用户线程对应一个内核线程。

优点是线程之间的切换由用户态的代码来进行，线程切换速度要快。对用户线程的数量几乎无限制。缺点是一个用户线程阻塞，那么其它所有线程都将无法执行，因为此时内核线程也随之阻塞。另外一个缺点是，在多处理器上，处理器数量的增加对线程性能不会有明显的增加。

3> 多对多模型（N:M），N个用户线程对应M个内核线程。

结合了一对一模型和多对一模型的优点：<1>一个用户线程的阻塞不会导致所有线程的阻塞，因为此时还有别的内核线程可以被调度执行；<2>多对多模型对用户线程的数量没有限制；<3>在多处理器的操作系统中，多对多模型的线程也能得到一定的性能提升。

7.2 goroutine

goroutine是Go语言中的轻量级线程实现，实现了N：M的线程模型，由Go运行时（runtime）管理。与传统的系统级线程和进程相比，其最大优势在于其“轻量级”，因为goroutine使用的是动态栈，可以很小到几K，所以，在一台服务器上可以轻松创建上百万个goroutine而不会导致系统资源衰竭，而一般的线程和进程通常也不会超过1万个。

goroutine的实现非常简单，普通函数前加上go关键字。这样，该函数会在一个新的goroutine中执行。分析这段代码的所有可能执行结果？

```
func main() {
    go fmt.Println("Hello")
    fmt.Println("World ")
}
```

7.3 channel

Go语言提倡“通过消息通信来共享内存，不要通过共享内存来通信”。所以在Go语言中，优先使用消息通信。channel是goroutine之间的消息通信机制。

首先看以下channel的基本操作。channel是类型相关的，一个channel只能传递一种类型的值，这个类型在创建channel时指定。

1 创建channel

使用内置的make函数创建一个channel。

```
chi := make(chan int)
chs := make(chan string)
chf := make(chan interface{})
```

7.3 channel

和map类似，channel也是一个对应make创建的底层数据结构的引用。当我们复制一个channel或用于函数的参数传递时，我们只是拷贝了一个channel引用。channel的零值也是nil。

两个相同类型的channel可以使用==运算符比较。如果两个channel引用的是相通的对象，那么比较的结果为真。一个channel也可以和nil进行比较。

一个channel有发送和接受两个主要操作，都是使用<-运算符。一个不保存接收结果的接收操作也是合法的。

```
ch<-x    //发送
x=<-ch   //接收
<-ch     //接收操作，但不保存接收结果
```

7.3 channel

channel还支持close操作，用于关闭channel，随后对该channel的任何发送操作都将导致panic异常。对一个已经关闭的channel接收数据，依然可以接收到已经成功发送的数据，如果没有的话，将接收到零值的数据。使用内置的close函数可以关闭channel。

```
close(ch)
```

7.3 channel

1 无缓存的channel

一个基于无缓存的channel的发送操作将导致发送者goroutine阻塞，直到另一个goroutine在相同的channel上执行接收操作。同样，如果接收操作先发生，那么接收者gotoutine也将阻塞，直到有另一个goroutine在相同的channel上执行发送操作。基于无缓存channel的发送和接收操作将导致两个goroutine做一次同步操作，因此，无缓存的channel也称为同步channel。

7.3 channel

2 串联的channel

channel可以将多个goroutine链接在一起，一个channel的输出作为下一个channel的输入。类似与进程间通信的管道。下面使用两个channel串联三个goroutine。

第一个gorutine用于生成0、1、2、3.....整数序列，通过channel传递给第二个goroutine；第二个goroutine将收到的整数求平方，然后将结果通过第二个channel传递给第三个goroutine，第三个goroutine打印收到的每个结果。

3 单向的channel

使用channel作为参数进行通信。在函数内部，有的channel只接收数据，有的channel只发送数据，这时可以使用单向的channel来表达这种意图。箭头<-和关键字chan的相对位置表明了channel的方向。

chan<-int 表示只向channel发送不从channel接收；

<-chan int 只从channel接收不向channel发送。

因为关闭操作用于确定不再向channel发送新的数据，所有只有在发送者所在的goroutine才会调用close函数，因此对一个只接受数据的channel调用close函数将是一个编译错误。

任何双向的channel向单向的channel赋值都将导致隐式转换。但是不能反向转换。

4 带缓存的channel

带缓存的channel内部持有一个元素队列。队列的最大容量是在调用make函数创建channel时，通过第二个参数指定的。例如：

```
ch = make(chan string,3)
```

向缓存channel的发送操作就是向内部缓存队列的尾部插入元素，接收操作则是从队列的头部删除元素。如果内部缓存队列是满的，发送操作将阻塞。如果channel是空的，接受操作将阻塞。通过缓存队列解耦了接收和发送的goroutine。

cap函数可以获取channel内部缓存的容量。len函数可以获取channel内部缓存队列中有效元素的个数。多个goroutine并发的向同一个channel发送数据或从同一个channel接收数据都是常见的用法。

5 并发的循环

并发的循环指的是在循环体内，通过go+匿名函数生成多个goroutine，如果在goroutine内用到外部函数的变量，不要直接使用，需要将外部变量作为匿名函数的参数传递，保证每个gotoutine运行不同的变量。

```
func loopgo() {
        for i := 1; i < 10; i++ {
                go func() {
                        fmt.Printf("第%d 个 goroutine\n", i)
                }()
        }
}
```

```
func loopgo() {
        for i := 1; i < 10; i++ {
                go func(i int) {
                        fmt.Printf("第%d 个 goroutine\n", i)
                }(i)
        }
}
```

6 基于select的多路复用

Go语言直接在语言级别支持select关键字，用于处理异步IO问题 。基于此特性，我们来为channel实现超时机制

```
select {
case <-chan1:
// 如果 chan1 成功读到数据，则进行该 case 处理语句
case chan2 <- 1:
// 如果成功向 chan2 写入数据，则进行该 case 处理语句
default:
// 如果上面都没有成功，则进入 default 处理流程
}
```

7 并发的退出

当一个已经关闭的channel中已经发送的数据都被成功接收后，后续的接收操作将不再阻塞，它们会立即返回一个零值。可以将这个机制扩展作为广播机制:不要向channel发送值，而是用关闭一个channel来广播。

7.4 共享内存

goroutine之间的的另外一种通信方式是共享内存，也就是访问相同的数据。但是只要两个或两个以上的goroutine同时访问数据，并且至少有一个是写操作时，就会发生数据竞争。解决数据竞争的方式是采用锁机制。在Go语言中主要通过sync包实现。

1 sync.WaitGroup

WaitGroup可称为组等待，可以阻塞main goroutine的执行，直到所有其它的一组goroutine执行完成。WaitGroup有三个方法，作用如下：

```
// 计数器增加 delta，delta 可以是负数。
func (wg *WaitGroup) Add(delta int)
// 计数器减少 1
func (wg *WaitGroup) Done()
// 等待直到计数器归零。如果计数器小于 0，则该操作会引发 panic。
func (wg *WaitGroup) Wait()
```

2 sync.Mutex互斥锁

Mutex为互斥锁，Lock()加锁，Unlock()解锁。使用Lock()加锁后，不能再次对其进行加锁，直到利用Unlock()解锁对其解锁后，才能再次加锁。如果在使用Unlock()前未加锁，将引起一个panic异常。Mutex并不与特定的goroutine相关联，可以在一个goroutine中加锁，在另一个goroutine中解锁。相关方法如下：

```
// Lock 用于锁住 m，如果 m 已经被加锁，则 Lock 将被阻塞，直到 m 被解锁。
func (m *Mutex) Lock()
// Unlock 用于解锁 m，如果 m 未加锁，则该操作会引发 panic。
func (m *Mutex) Unlock()
```

3 sync.RWMutex读写锁

读写锁控制下的多个写操作之间都是互斥的，并且写操作与读操作之间也都是互斥的。但是，多个读操作之间却不存在互斥关系。常用于读次数远远多于写次数的场景，称为多读单写锁。

```
// Lock 将 rw 设置为写锁定状态，禁止其他例程读取或写入。
func (rw *RWMutex) Lock()
// Unlock 解除 rw 的写锁定状态，如果 rw 未被写锁定，则该操作会引发 panic。
func (rw *RWMutex) Unlock()
// RLock 将 rw 设置为读锁定状态，禁止其他例程写入，但可以读取。
func (rw *RWMutex) RLock()
// Runlock 解除 rw 的读锁定状态，如果 rw 未被读锁定，则该操作会引发 panic。
func (rw *RWMutex) RUnlock()
```

4 sync.Once初始化

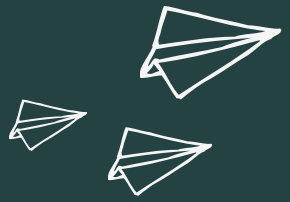
Once 的作用是多次调用但只执行一次，Once 只有一个方法，Once.Do()，向 Do 传入一个函数，这个函数在第一次执行 Once.Do() 的时候会被调用，以后再执行 Once.Do() 将没有任何动作。

```
// 多次调用仅执行一次指定的函数 f
func (o *Once) Do(f func())
```

5 竞争条件检测

即使我们小心到不能再小心，但并发程序中出错还是太容易了。幸运的是，Go的runtime和工具链为我们装备了一个复杂但是好用的动态分析工具，竞争检查器。使用时，只需要在go build，go run或者go test命令后加上-race。完整的文档在The Go Memory Model文档中有说明，和语言文档放在一起。

https://golang.org/ref/mem



THANKS

感谢聆听，期待反馈